1 Introduction

Résumé en français : Nous nous intéressons ici à la longueur palindromique préfixale $PPL_w(n)$ d'un mot w, c'est-à-dire le nombre minimal de palindromes concaténés nécessaires pour obtenir le préfixe de longueur n. Divers résultats ont déjà été obtenus, notamment sur l'aspect non borné de la suite $(PPL_u(n))_{n\geq 0}$ pour la plupart des mots non-ultimement périodiques [1], ou encore un calcul précis de cette suite pour le mot de Thue-Morse en utilisant la suite des différences palindromiques d définie par $d(n) = PPL(n+1) - PPL(n)$. Cette suite est définie sur l'alphabet $\{+1, 0, -1\}$, et dans le cadre du mot de Thue-Morse elle se construit par le morphisme suivant [2] :

$$\delta = \begin{cases} +1 & = & +1 & +1 & 0 & -1 \\ 0 & = & +1 & +1 & -1 & -1 \\ -1 & = & +1 & 0 & -1 & -1 \end{cases}$$

Dans cette version du document est détaillé le fonctionnement du programme python `AStools.py` calculant la suite $PPL(n)$ pour un mot w (en utilisant la structure `eertree` [3]), son k-noyau, sa complexité et cherchant si la suite d peut être définie par un morphisme uniforme. Les résultats du programme sur ce dernier point auront beau n'être que calculatoires, leur validité sera renforcée par l'étude du noyau de la suite d, de sa complexité, ainsi que par la validité des résultats sur le mot de Thue-Morse. Notamment, dans un papier de E. Charlier, N. Rampersad et J.O. Shallit [5], il a été montré que diverses fonctions sur les mots k-automatiques sont k-régulières, mais la fonction PPL semble faire exception, pour le mot du doublage de période.

English abstract : We are looking at the prefix palindromic length of a infinite word w, denoted $PPL_w(n)$, as the minimal number of concatenated palindromes needed to express the prefix of length n of w. Some results have already been proved like the unbounded aspect of $PPL(n)$ for almost all non-ultimately periodic word [1], or even the precise computation of the palindromic diference $d(n) = PPL(n+1) - PPL(n)$ for the Thue-Morse word [2]. This sequence d, defined on the alphabet $\{+1, 0, -1\}$ is the fixed point of the morphism above, in the Thue-Morse word case.
In this document, we describe the python program `AStools.py` that compute the PPL sequence (using `eertree` structure [3]), the k-noyau, the complexity, and determine if the infinite sequence d can be obtained as a fixed point of a uniform morphism. Even if the result is only computational, their validity are reinforced by the noyau and complexity computation, as well as the validity of the results on the Thue-Morse word. In particular, in a paper by E. Charlier, N. Rampersad and JO Shallit [5], it has been shown that various functions on k-automatic words are k-regular, but the PPL function seems to be an exception for the period doubling word.

Cette version regroupe seulement les sections utiles au programme `AStools.py`

2 Étude de suites

Soit w un mot, on notera son i-ème caractère par $w[i]$, et le sous-mot de w composé des caractères $i+1$ à j par $w(i..j]$. On a donc $w = w(0..n]$, pour w de longueur n. Un palindrome est un mot qui se lit de la même manière à l'envers et à l'endroit, c'est donc un mot p de longueur n tel que $p[i] = p[n+1-i]$, pour tout i. Pour un alphabet A et $a \in A$, on notera a^n la concaténation de n lettres a, A^* les mots de longueur finies sur A et A^ω les mots infinis. On s'intéresse à la longueur palindromique du préfixe de taille n d'un mot w, notée $PPL_w(n)$. Il s'agit du nombre minimal de palindromes nécessaires pour décomposer $w(0..n]$.
Par exemple, le mot *ababbba* est décomposable en 3 palindromes minimum : $(aba)(bbb)(a)$, ainsi si un mot w commence par *ababbba*, on aurait $PPL_w(7) = 3$.

Dans la suite, on utilisera indifféremment *mot* et *suite*, une mot étant une suite de caractères, et une suite étant une liste de caractères que l'on peut concaténer pour obtenir un mot. On va maintenant donner diverses définitions et propriétés nécessaires, détaillées dans [4], avant d'introduire la définition de suite k-automatique :

Définition 1 - DFAO Un automate fini déterministe avec sortie, ou DFAO pour l'appellation anglaise *deterministic finite automaton with output* est la donnée d'un automate $\mathcal{M} = (\mathcal{Q}, \Sigma, q_0, \delta, A, \tau)$, où $\mathcal{Q}$ est l'ensemble des états, Σ est l'alphabet de w, q_0 est l'état initial, δ est la fonction de transition, mais sans ensemble d'états finaux F et avec un alphabet de sortie, noté A, et une fonction $\tau : \mathcal{Q} \to A$ telle que pour chaque état q dans lequel l'automate s'arrête en lisant un mot w on ait un caractère de sortie dans A qui y soit associé.

Soit n un nombre dans une base quelconque, on note alors $(n)_k$ son écriture en base k.

Définition 2 - Suite k-automatique On dit qu'un mot infini $w = w[1] \cdot w[2] \cdot w[3] \cdots$ sur un alphabet A est k-automatique s'il existe un DFAO $\mathcal{M}$ tel que $w_i = \tau(\delta(q_0, (i)_k)), \forall i \geq 0$.

Dans la suite, on n'utilisera pas cette définition de suite k-automatique, mais une définition équivalente utilisant les morphismes k-uniforme [4].

Définition 3 - Morphisme k-uniforme Un morphisme est une fonction $\phi : A^* \to B^*$ qui satisfait $\phi(xy) = \phi(x)\phi(y)$. On suppose ici que $A = B$ et qu'il existe un $k \geq 2$ tel que pour tout $a \in A$, $|\phi(a)| = k$, on dit que ϕ est un morphisme k-uniforme :

$$\exists k \geq 2, \forall a \in A, |\phi(a)| = k$$

S'il existe un caractère $a \in A$ tel que $\phi(a) = ax$ pour $x \in A^*$ de longueur $k-1$, alors ϕ est dite prolongeable en a et dans ce cas, le mot infini $w = \phi^\omega(a) := ax\phi(x)\phi^2(x)\cdots$ est l'unique point fixe infini de ϕ commençant par a. On étend naturellement la notion de morphisme k-uniforme aux mots infinis A^ω.

On rappelle qu'un codage est un morphisme $\nu : A \to B$ qui pour chaque caractère de A, associe un caractère de B. On peut naturellement étendre la notion de codage sur les mots (et mots infinis) de A vers B car ν est un morphisme.

Théorème 4 - Équivalence automate et morphisme k-uniforme Soient A et B deux alphabets. Soit $k \geq 2$, un mot $w = w[1] \cdot w[2] \cdot w[3] \cdots \in B^\omega$ est k-automatique si et seulement si w est l'image, par un codage $\nu : A^\omega \to B^\omega$, d'un point fixe $u \in A^\omega$ par un morphisme k-uniforme $\phi : A^\omega \to A^\omega$, i.e. :

$$\text{Soient } \phi : A^\omega \to A^\omega, \nu : A^\omega \to B^\omega \ : \quad w = \nu(u) \quad \text{où} \quad u = \phi(u)$$

Exemple 5 - Mot de Thue-Morse Par exemple, le mot de Thue-Morse t est définit par le morphisme 2-uniforme $\phi : a \mapsto ab, b \mapsto ba$ et par le codage trivial $\nu(x) = x$, $x \in A$, pour $A = \{a, b\}$ (donc $A = B$). On obtient :

$$t = abba\ baab\ baab\ abba\ baab\ abba\ abba\ baab\ \ldots$$

Définition 8 - k-noyau On définit le k-noyau d'un mot infini w comme étant l'ensemble des sous-séquences :

$$K_k(w) = \{(w_{k^i \cdot n + j})_{n \geq 0} : i \geq 0 \text{ et } 0 \leq j < k^i\}$$

On appellera $(i+1)$-ème itération du k-noyau l'ensemble des sous-suites ajoutées en prenant seulement un terme sur k^l, pour $l \leq i$, la 1ère itération étant celle avec $i = 0$:

$$K_k^i(w) = \{(w_{k^i \cdot n + j})_{n \geq 0} : 0 \leq j < k^i\}$$

3 Programme Python

Le programme python `AStools.py` [6] génère un mot k-automatique w et calcule ses suites PPL_w et d_w associés, puis permet de déeterminer le noyau ou de la complexité de w ou d_w, ainsi que la détection d'un morphisme k-automatique générant la suite d_w. Pour les calculs de PPL et d, il utilise la structure `eertree` [3].

3.1 Explications des parties du programme

Voici des explications sur certaines parties principales :

Morphism detect : La fonction `morphismDetect` détermine si le préfixe de longueur n du mot w peut être généré à l'aide d'un morphisme. Pour se faire elle utilise les variables `wordSize` et `mapSize` qui déterminent la taille du mot d'entrée et le nombre de fois que l'image est plus grande que l'antécédent. *(1) Tant que* la taille de l'antécédent n'est pas maximale (`wordSizeMax`), on augmente sa taille de $+1$ et on itère sur la taille de l'image. *(2) Tant que* la taille de l'image n'est pas maximale (`wordSizeMax * mapSizeMax`), on augmente sa taille de `+wordSize` puis on regarde si le morphisme qui s'en déduit est valide, i.e. : *(3) Tant que* l'on peut lire l'antécédent et l'image et qu'il n'y a pas d'erreur, on les stocke respectivement dans `c` et `d` et on vérifie *si* `c` n'est jamais apparu. Dans ce cas là on vérifie que le morphisme sera bijectif et *si* c'est le cas : on met à jour le morphisme et on vérifie que toute les occurances de `c` ont la même image (`checkMorphismEntry()`). Si `c` a une seule image, alors on reprend à l'étape *(3)* avec le mot `c` suivant, sinon il y a une erreur et on retourne à l'étape *(2)*.
Si on sort de *(3)* sans obtenir d'erreur, alors le morphisme est valide, on sort de *(1)* et l'on renvoie le morphisme et [le nombre d'occurances des antécédents/le codage] en fonction des arguments. Sinon on renvoie deux dictionnaires vides.

`checkMorphismEntry(w,c,d,s,m)` **:** Cette fonction renvoie le nombre d'occurences de `c` qui ont la même image que `d` dans la séquence `w`. Si une image n'est pas égale à `d`, alors elle renvoie le nombre précédent multiplié par `-1`. Cette vérification est effectuée au plus `m` fois, à parir de la position `s`.

Kernel : La fonction `noyauIter(C,u,i,k,S)` prends l'ancienne itération `C` du k-noyau, calcule la `i+1`-ème itération du k-noyau ($k^i \times n + j$) de la séquence u et retourne le résultat. Toute les suites dans `C` sont de taille `S`.

3.2 Exemples commentés

python `AStools.py ab,ba -n128 -w`
Calcul les 128 premier termes de la suite w de Thue-Morse `[-n128]`, et les affiche `[-w]`.

python `AStools.py ab,ba -p32 -k2d -i12 -s64 -z2 -IShow -IVerb -m`
Cette commande génère les suites PPL et $d(n)$ et affiche les 32 premiers termes de PPL `[-p32]`. Elle calcule ensuite le 2-noyau de la suite $d(n)$ `[-k2d]` jusqu'à 12 itérations (2^{12}) `[-i12]`, et toutes les suites du noyau sont de longueur 64 `[-s64]`. Elle multiplie ensuite 2 fois la taille de la suite par k `[-z2]` et re-calcule le noyau. Enfin, elle affiche le contenu final du noyau `[-IShow]`, affiche les noyaux intermédiaires `[-IVerb]` et affiche les résultats sous forme de matrice `[-m]`.

python AStools.py ab,cb,ad,cd 0,0,1,1 -n1048576 -p16 -f -FWord64 -FMap2
Le programme génère le mot $w = abcbadcbabcd\dots$ de longueur 1048576, puis applique le codage $a, b \mapsto 0$ et $c, d \mapsto 1$ [ab,cb,ad,cd 0,0,1,1] et affiche les 16 premiers termes de $PPL(n)$. Ensuite il cherche si la suite $d(n)$ est définie par un morphisme [-f], en considérant au plus des mots de taille 64 [-FWord64], et en cherchant un morphisme au plus 2-uniforme [-FMap2]). Ici le nombre de vérification [-FStep] n'est pas précisé, et vaut donc 50.

python AStools.py aba,bbb -n729 -d32 -c10d
Génère les 729 premiers termes de la suite de Sierpinski [-n729], calcule la suite d et affiche ses 32 premiers termes [-d32], puis calcul et affiche le nombre de mots de taille 1 à 10 dans la suite d [-c10d] (la complexité).

3.3 Utilisation et liste des arguments

Commande pour appeler le programme dans un terminal :

python AStools.py <arguments>

Le programme prenant différents arguments, ils sont listés ci-dessous avec leur valeur par défaut, ainsi que leur description :

Syntaxe	Défaut	Description
Arguments à position fixe		
[morphisme]	-	**Obligatoire** et toujours en première place, spécifie le morphisme utilisé pour construire le mot. Syntaxe : ab,cb,ad,cd pour le mot du *pliage de papier*.
[codage]	-	**Facultatif** et toujours en seconde place, applique un codage si nécessaire. Syntaxe : 0,0,1,1 pour le mot du *pliage de papier* enverra a et b sur 0, et c et d sur 1.
Arguments principaux		
-n<val>	2048	Spécifie la longueur du mot à Calculer. Par exemple, -n64 génèrera les 64 premiers termes.
-w<val>	-n	Affiche les val premiers termes du mot généré. Si aucune valeur n'est spécifiée, affiche intégralement le mot.
-p<val>	-n	Affiche les val premiers termes de la longueur palindromique du préfixe du mot w. Si aucune valeur n'est spécifiée, affiche toute la suite.
-d<val>	-n	Affiche les val premiers termes de la suite $d(n)$ du mot w. Si aucune valeur n'est spécifiée, affiche toute la suite.
Pour le calcul de complexité		
-c<val><seq>	10w	Calcul la complexité $c(n)$ jusqu'aux mots de taille val dans la suite seq.
-CStep<val>	1	Calcul la complexité val fois pour les préfixes de longueur entre -n/val et -n inclu.

Syntaxe	Défaut	Description
Pour le calcul du noyau		
-k<val><seq>	2w	Calcul le val-noyau de la suite seq. val doit être un entier et seq parmi w, p ou d Par exemple -k3d Calculra le 3-noyau de la suite $d(n)$. Ne pas utiliser -n avec -k. La longueur est automatiquement calculée pour donner des valeurs justes.
-i<val>	6	Itérations maximale à effectuer pour Calculr le noyau.
-s<val>	32	Longueur des suites dans le noyau.
-z<val>	0	Itérations du calcul du noyau sur des longueurs croissantes du préfixe du mot considéré. Par exemple, -k2d -z3 Calculra le noyau de $d(n)$, 4 fois, en multipliant à chaque fois la longueur du préfixe par k.val.
-u<val>	-s	Destiné à l'**affichage** uniquement, affiche seulement les val premiers termes des suites dans le noyau.
-IShow<val>	0 ou -i	Affiche le contenu de l'itération i du noyau. Si l'argument n'est pas utilisé, n'affiche pas le contenu. Si aucune valeur n'est donnée, affiche le contenu du dernier noyau. Si val = -1, affiche le contenu de tous les noyaus.
-IVerb	*False*	Affiche toutes les itérations intermédiaires calculées du noyau (pour k^j, $j < i - 1$).
-m	*False*	Affiche la matrice de cardinalité du noyau (utile avec -i et -z).
Pour la détection de morphisme		
-f	-	Active la recherche d'un morphisme définissant la suite $d(n)$.
-FStep<val>	50	Nombre de vérifications (recherche) pour valider le morphisme. Si l'argument n'est pas utilisé, prends la valeur par défaut.
-FWord<val>	4	Taille maximale du mot d'entrée du morphisme. Si l'argument n'est pas utilisé, prends la valeur par défaut.
-FMap<val>	10	Taille maximale de l'image du morphisme (Morphisme au plus FMap-uniforme). Si l'argument n'est pas utilisé, prends la valeur par défaut.
-FCode	*False*	Si le morphisme détecté prend des mots de taille 2 ou plus, et s'il a au plus 62 antécédents, renvoyer un morphisme et un codage plutot qu'un morphisme sur l'alphabet $\{+, 0, -\}$.
Développement		
--args	-	Affiche le dictionnaire contenant tous les arguments utilisés ci-dessus.

4 Résultats obtenus

4.1 Le mot de Thue-Morse

Le premier test à effectuer avec ce programme est de vérifier s'il donne des résultats valides, en lui demandant notamment de déterminer lui-même le morphisme définissant la suite $d(n)$ du mot de Thue-Morse, détaillé dans l'introduction. Pour ce faire, on entre la commande ci-dessous dans un terminal : `python AStools.py ab,ba -n4096 -d32 -f -FWord1 -FMap4` .

```
Morphism found: {'+': '++0-', '0': '++--', '-': '+0--'}

Antecedent (3): + , 0 , -
```

Les dernières lignes affichées par le programme disent que la suite $d(n)$ de Thue-Morse peut être définie comme étant le point fixe du morphisme `+` $\mapsto$ `++0-` ; `0` $\mapsto$ `++--` ; `-` $\mapsto$ `+0--`, et il s'agit bel et bien du morphisme explicité en introduction [2].

4.2 Sur d'autres mots infinis

Dans la suite du document et pour éviter de charger l'écriture, pour un mot w, on notera φ au lieu de φ_w le morphisme générant la suite d_w, et ν au lieu de ν_w son codage. S'il y a ambiguïté, on précisera en rajoutant un indice. On mettra également en page les résultats pour plus de lisibilité. Enfin, dans cette section les codages des mots d_w sont vus comme des morphismes l-uniforme car il permettent de simplifier l'écriture des résultats.

Le mot de Rudin-Shapiro

Après la vérification sur le mot de Thue-Morse, on peut obtenir des résultats pour d'autres mots infinis. Par exemple sur le mot de Rudin-Shapiro $r = 0001001000011101\cdots$, défini par le morphisme $\phi : a \mapsto ab, b \mapsto ac, c \mapsto db, d \mapsto dc$ et par le codage $\tau : a, b \mapsto 0\ ,\ c, d \mapsto 1$, on obtient le mot d_r suivant : `+00+00000-++00-++00-+0+00+00+00+` $\cdots$, et la recherche par le programme d'un morphisme uniforme définissant cette suite donne les résultats suivants :

$$d_r = \nu(\varphi^\omega(a)) \quad \text{où} \quad \varphi = \begin{cases} a \mapsto ab \\ b \mapsto cd \\ c \mapsto eb \\ d \mapsto ed \\ e \mapsto cb \end{cases} \quad \text{et} \quad \nu \text{ définit par :}$$

a = +00+00000−++00−++00−+0+00+00+00+−0+00−+00+−0+0+0−0+00−+00+00+00+
b = 0+0−0++−00+0+0−++00−+0+00+00+00+0+0−0++−00+0+0−+000+0−+00+00+00+
c = −0+00−+00+−0+0+00+00−++−+00+000+0−+000+−+0−0+0+0−0+00−+00+00+00+
d = −0+00−+00+−0+0+00+00−++−+00+000+0+0−0++−00+0+0−+000+0−+00+00+00+
e = 0+0−0++−00+0+0−++00−+0+00+00+00+−0+00−+00+−0+0+0−0+00−+00+00+00+

On obtient un morphisme φ 2-uniforme, et un codage ν 64-uniforme. Le mot d_r est ainsi obtenu comme point fixe de $\nu(\varphi^\omega(a))$. Ces résultats sont obtenus par la commande suivante `python AStools.py ab,ac,db,dc 0,0,1,1 -n64000 -d0 -f -FWord64 -FMap2 -FStep100`. De plus, le 2-noyau de la suite d_r semble fini, de cardinalité $Card(K_2(d_r)) = 78$. Plus aucune nouvelle suite n'apparait dans celui-ci à partir de la 8e itération du calcul du noyau (itération $2^7 \times n + j$).
On sait que dans r, le plus grand palindrome est de longueur 14, de ce fait, $\{n : PPL_r(n) = 1\}$ est fini : On trouve en effet que $PPL_r(n) = 1$ seulement pour $n = 1, 2, 3$ et 10. Si l'on définit $\mathcal{V}(x)$ comme le nombre de caractères `+` moins le nombre de `-` dans x, pour x un codage de ν, alors $\mathcal{V}(a) = 21 - 8 = 13$, et de manière générale, pour $x \in \{a, b, c, d, e\}$, $\mathcal{V}(x) > 0$: $\mathcal{V}(b) = 15$, $\mathcal{V}(c) = 11$, $\mathcal{V}(d) = \mathcal{V}(e) = 13$.

Le mot du Pliage de Papier

Sur le mot du Pliage de papier $p = 0010011000110110\cdots$, défini par le morphisme $\phi : a \mapsto ab, b \mapsto cb, c \mapsto ad, d \mapsto cd$ et par le codage $\tau : a, b \mapsto 0\ ,\ c, d \mapsto 1$, on obtient le mot d_p suivant : `+0+0-+0+000-++0-+-0+00+00-0++-0+` $\cdots$, et la même recherche que précédemment donne :

$$d_p = \nu(\varphi^\omega(a)) \quad \text{où} \quad \varphi = \begin{cases} a \mapsto ab & f \mapsto ch \\ b \mapsto cd & g \mapsto if \\ c \mapsto ef & h \mapsto gh \\ d \mapsto gd & i \mapsto ib \\ e \mapsto eb & \end{cases} \quad \text{et} \quad \nu \text{ définit par :}$$

a	=	+0+0−+0+000−++0−+−0+00+00−0++−0+000+00+0−+000+000+00000−0++0−+0−
b	=	0++−0+00+00−0+00000+00+00−0++−0+0+−+0−0+000+000+0+−+0−000+000+0−
c	=	0++−0+00+00−0+00000+00+00−0++−0+000+00+0−+000+000+00000−0++0−+00
d	=	+00+−00+0000+0000−0+00+00−0++−0+0+−+0−0+000+000+0+−+0−000+000+0−
e	=	0++−0+00+00−0+00000+00+00−0++−0+000+00+0−+000+000+00000−0++0−+0−
f	=	0++−0+00+00−0+00000+00+00−0++−0+0+−+0−0+000+000+0+−+0−000+000+00
g	=	0+00000+00000+000−0+00+00−0++−0+000+00+0−+000+000+00000−0++0−+00
h	=	+00+−00+0000+0000−0+00+00−0++−0+0+−+0−0+000+000+0+−+0−000+000+00
i	=	0+00000+00000+000−0+00+00−0++−0+000+00+0−+000+000+00000−0++0−+0−

Le morphisme φ obtenu est également 2-uniforme, et le codage ν est 64-uniforme. Le mot d_p est donc le point fixe de $\nu(\varphi^\omega(a))$. Et le 2-noyau de la suite d_p semble également fini, de cardinalité $Card(K_2(d_p)) = 68$, où plus aucune nouvelle suite n'apparait dans celui-ci à partir de la 8e itération du calcul du noyau (itération $2^7 \times n + j$).
Tout comme le mot r, le nombre de palindromes dans p est fini, et soit $\mathcal{V}$ tel que définit précedemment : $\{n : PPL_p(n) = 1\} = \{1, 2, 5\}$ car $\mathcal{V}(x) > 0$, pour $x \in \{a, b, c, d, e, f, g, h, i\}$: $\mathcal{V}(a) = \mathcal{V}(b) = \mathcal{V}(e) = \mathcal{V}(g) = \mathcal{V}(h) = 10$, $\mathcal{V}(c) = \mathcal{V}(f) = 11$ et $\mathcal{V}(d) = \mathcal{V}(i) = 9$.

Le mot de Sierpinski

Sur le mot de Sierpinski $s = aba\ b^3\ aba\ b^9\ aba\ b^3\ aba\ b^{27}\cdots$, défini par le morphisme $\phi : a \mapsto aba, b \mapsto bbb$, on obtient le mot d_s suivant : `++-+00+--+00000000+0000-+--` $\cdots$, et les résultats suivants :

$$d_s = \nu(\varphi^\omega(a)) \quad \text{où} \quad \varphi = \begin{cases} a \mapsto abc & f \mapsto ddf \\ b \mapsto bdd & g \mapsto ifg \\ c \mapsto efg & h \mapsto edj \\ d \mapsto ddd & i \mapsto abj \\ e \mapsto edh & j \mapsto hdj \end{cases} \quad \text{et} \quad \nu \text{ définit par :}$$

a	=	++−+00+−−	;	b	=	+00000000	;	c	=	+0000−+−−
d	=	000000000	;	e	=	+00000+0−	;	f	=	00000000−
g	=	++−00−+−−	;	h	=	+0000000−	;	i	=	++−+0000−
j	=	+0−00000−								

Ce morphisme φ est 3-uniforme, et le codage ν est 9-uniforme. Le mot d_s est le point fixe de $\nu(\varphi^\omega(a))$ tels que définis ci-dessus. De plus, le 3-noyau de la suite d_s semble fini, de cardinalité $Card(K_3(d_s)) = 23$, et sa complexité semble linéaire, bornée (de manière non-optimale) par $c_{d_s}(n) < 10n$.

Ci-dessous sont détaillés les observations sur le mot d_s, on donne le tableau suivant représentant la longueur palindromique de son préfixe de taille n :

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
s(n)	a	b	a	b	b	b	a	b	a	b	b	b	b	b	b	b	b	b
$PPL_s(n)$	1	2	1	2	2	2	3	2	1	2	2	2	2	2	2	2	2	2
$d_s(n-1)$	+	+	-	+	0	0	+	-	-	+	0	0	0	0	0	0	0	0

n	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
s(n)	a	b	a	b	b	b	a	b	a	b	b	b	b	b	b	b	b	b
$PPL_s(n)$	3	3	3	3	3	2	3	2	1	2	2	2	2	2	2	2	2	2
$d_s(n-1)$	+	0	0	0	0	-	+	-	-	+	0	0	0	0	0	0	0	0

On remarque que la construction du mot de *Sierpinski* permet de deviner facilement une grande partie de la suite d_s, celle-ci étant divisée en trois. Prenons le mot de longueur 3^n, formé par $\phi_s^n(a)$, alors la première partie est celle du préfixe de longueur 3^{n-1} de d_s, la deuxième partie est le mot `+0···0` de longueur 3^{n-1}, et la dernière partie semble être définie par rapport aux termes précédents. Mais on remarque également un détail dans le morphisme générant d_s :

Soient l'opération commutative $\oplus$ sur l'alphabet $\{-, 0, +\}$ et le codage suivant

$$c : + \mapsto +1, \quad 0 \mapsto 0, \quad - \mapsto -1.$$

On définit $\oplus$, pour x et y dans $\{-, 0, +\}$, par :

$$x \oplus y = c^{-1}(min(1, max(-1, (c(x) + c(y)))))$$

En particulier, $+ \oplus + = +$ et $- \oplus - = -$. Soient maintenant u et u' des mots de même longueur n définis sur l'alphabet $\{-, 0, +\}$, on définit l'opération $\oplus$ sur de tels mots par :

$$u \oplus u' = v \quad \text{où} \quad v[i] = u[i] \oplus u'[n - i + 1], \qquad i \in \{1, ..., n\}$$

Et si les mots sont de longueur 1, on retourne sur la définition précédente.
Alors on peut grouper les codage de ν_s ci-dessous par deux de telle sorte que $x \oplus y = 0^n$, et dans ce cas on le notera $[x \oplus y]$. Les groupes sont donc $[a \oplus g]$,$[b \oplus f]$,$[c \oplus i]$,$[d \oplus d]$,$[e \oplus j]$ et $[h \oplus h]$, avec :

a = ++−+00+−−	;	b = +00000000	;	c = +0000−+−−
d = 000000000	;	e = +00000+0−	;	f = 00000000−
g = ++−00−+−−	;	h = +0000000−	;	i = ++−+0000−
j = +0−00000−				

Par exemple : `a` $\oplus$ `g` = `++-+00+--` $\oplus$ `++-00-+--` = `000000000`.
On peut aussi remarquer que la fin du mot $d_s(0..3^n]$ est ifg =`++-+0000-00000000-++-00-+-`, et également que $[abc \oplus ifg]$.

Le mot du doublage de période

Pour le mot du *doublage de période* défini par le morphisme $\phi : a \mapsto ab, b \mapsto aa$, le mot d_{dp} obtenu est `++-+00-++-+-+--+0+0-+00-+00-+--+`$\cdots$, mais aucun résultat n'a pu être trouvé quant au morphisme définissant ce mot.
Une étude de son 2-noyau montre que celui-ci ne semble pas être fini, on obtient en effet le tableau suivant, représentant la cardinalité du 2-noyau de $K_2(d_{dp})$:

préfixe \ iter.	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
32768	1	3	7	15	30	58	106	177	275	401	522
32768×2^1	1	3	7	15	30	60	111	193	326	487	671
32768×2^2	1	3	7	15	30	60	114	210	365	577	837
32768×2^3	1	3	7	15	30	60	116	218	399	657	1018
32768×2^4	1	3	7	15	30	60	118	224	415	719	1169
32768×2^5	1	3	7	15	30	60	118	226	423	755	1285
32768×2^6	1	3	7	15	30	60	118	226	425	768	1343

Les colonnes du tableau représentent le calcul du noyau : On peut dire que le noyau est fini si à partir d'une itération i, le noyau ne grandit plus. Passer d'une colonne 2^i à 2^{i+1} signifie ajouter les sous-suites de la forme $(dp[2^{i+1} \times n + j])_{n \geq 0}$ au noyau (voir **[définition 8]**).

Les lignes représentent les différentes longueurs du préfixe de dp : Plus le préfixe est grand et plus le calcul du noyau est précis car les sous-suites du noyau seront proportionnellement grandes. Calculer le noyau pour différentes longueurs de préfixe permet de voir si la cardinalité change avec la taille de celui-ci.

La complexité montre aussi que la suite d_s ne peut pas être automatique. Le tableau ci-dessous représente les complexités $c_{dp}(n)$ pour n entre 1 et 20 du préfixe de taille 5 millions, que l'on note $c_{dp(0..5\times10^6)}(n)$:

n	1	2	3	4	5	6	7	8	9	10	12	14	16	18	20
$c_{dp}(n)$	3	9	21	45	81	141	217	305	413	533	809	1109	1421	1747	2088

On peut y apercevoir qu'elle ne semble pas avoir de croissance linéaire :

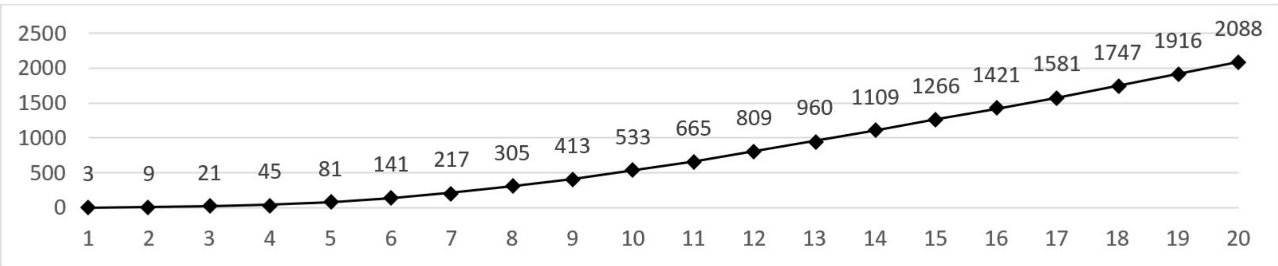

FIGURE 1 – Complexité de 1 à 20 du mot *dp*.

On note également que la complexité $c_{dp}(n)$ augmente lentement avec la taille du préfixe considéré, en effet certains mots de taille n n'apparaissent qu'après quelques millions de termes. Par exemple $c_{dp(0..2\times10^6]}(15) = 1251$, $c_{dp(0..3\times10^6]}(15) = 1256$ et $c_{dp(0..5\times10^6]}(15) = 1266$.

Le tableau suivant représente cette différence de complexité $c_s(n)$, $1 \leq n \leq 15$, pour les préfixes de taille 1 million à 5 millions :

n	$\cdots$	7	8	9	10	11	12	13	14	15
10^6	$\cdots$	217	301	405	520	645	779	918	1054	1191
2×10^6	$\cdots$	217	**305**	**413**	**533**	**664**	**806**	**954**	**1100**	**1251**
3×10^6	$\cdots$	217	305	413	533	664	806	**955**	**1102**	**1256**
4×10^6	$\cdots$	217	305	413	533	664	806	955	1102	1256
5×10^6	$\cdots$	217	305	413	533	**665**	**809**	**960**	**1109**	**1266**

5 Conclusion

En français : Ces nouveaux résultats calculatoires permettent de faire de nouvelles hypothèses, ils semblent notamment montrer un point commun entre le morphisme ϕ_w générant un mot w, et celui générant sa suite d_w, φ_w : si ϕ_w génère des palindromes, alors ils semblent tous deux k-automatiques, pour un même entier k. Par exemple pour le mot de Thue-Morse, le morphisme $\phi_t^2 : a \mapsto abba, b \mapsto baab$ est 4 automatique, tout comme d_t. Pour le mot de Sierpinski $\phi_s : a \mapsto aba, b \mapsto bbb$, les résultats précédents donnent un morphisme φ_s également 3-automatique. Pour un mot w non k-automatique, on peut donc s'attendre à ce que le morphisme φ_w ne soit pas k-automatique. Quant à la taille des antécédents du morphisme φ_w, elle semble plus compliquée à définir.
Conjecture : Soit k un entier et A un alphabet fini. Soit ϕ un morphisme k-uniforme sur A et τ un codage générant w un mot k-automatique. Si $\forall x \in A$, $\tau(\phi(x))$ est un palindrome, alors d_w est k-automatique.

In English : These new computational results allow us to make new hypotheses, they seem in particular to show a common point between the morphism ϕ_w generating a word w, and the one generating his d_w sequence, φ_w : if ϕ_w generates palindromes, so they both seems k-automatic, for the same integer k. For example for the Thue-Morse word, the morphism $\phi_t^2 : a \mapsto abba, b \mapsto baab$ is 4-automatic, just like his d_t sequence. For Sierpinski's word $\phi_s : a \mapsto aba, b \mapsto bbb$, the previous results give a morphism φ_s also 3-automatic. For a word w not k-automatic, we can expect the morphisme φ_w is not k-automatic. As for the size of the antecedents of the morphism φ_w, it seems more complicated to define.
Conjecture : Let k be an integer and A an alphabet. Let ϕ be a k-uniform morphism over A and τ a coding that generate a k-automatic word w. If $\forall x \in A$, $\tau(\phi(x))$ is a palindrome, then d_w is k-automatic.

Bibliographie

[1] A.E. Frid, S. Puzynina, L. Zamboni, On palindromic factorization of words, *Advances in Appl. Math.*, 2013, 737-748.

[2] A.E. Frid, Prefix palindromic length of the Thue-Morse word, 2019.

[3] M. Rubinchik, A. M. Shur, EERTREE : An Efficient Data Structure for Processing Palindromes in Strings, 2015.

[4] J.-P. Allouche, J.O. Shallit *Automatic sequences : theory, applications, generalizations*, 2003.

[5] E. Charlier, N. Rampersad, J.O. Shallit, *ENUMERATION AND DECIDABLE PROPERTIES OF AUTOMATIC SEQUENCES*, 2011.

[6] `AStools.py`, https://github.com/Enzo-Laborde/AutoSeqTools.